Theoretical Computer Science 134 (1994) 473–491
Elsevier

Approximating shortest superstrings with constraints

Tao Jiang[1]

Department of Computer Science, McMaster University, Hamilton, Ont., Canada L8S 4K1

Ming Li[2]

Department of Computer Science, University of Waterloo, Waterloo, Ont., Canada N2L 3G1

Communicated by O.H. Ibarra
Received September 1992
Revised March 1993

Abstract

Jiang, T. and M. Li, Approximating shortest superstrings with constraints, Theoretical Computer Science 134 (1994) 473–491.

Various versions of the shortest common superstring problem play important roles in data compression and DNA sequencing. Only recently, the open problem of how to approximate a shortest superstring given a set of strings was solved in (Blum et al., 1991; Li, 1990). Blum et al. (1991) shows that several greedy algorithms produce a superstring of length $O(n)$, where n is the optimal length. However, a major problem remains open: can we still linearly approximate a superstring in polynomial time when the superstring is required to be consistent with some given negative strings, i.e., it must not contain any negative string? The best previous algorithm, Group-Merge given in (Jiang and Li, 1993; Li, 1990), produces a consistent superstring of length $\theta(n \log n)$. The negative strings make the problem much more difficult and, as we will show, a greedy-style algorithm cannot achieve linear approximation for this problem.

We present polynomial-time approximation algorithms that produce consistent superstrings of length $O(n)$, for two important special cases: (a) when no negative strings contain positive strings as substrings; (b) when there are only a constant number of negative strings. The algorithms are obtained by making an essential use of the Hungarian algorithm, which can find an optimal cycle cover on weighted graphs.

The other main objective of this paper is to analyze the performance of some greedy-style algorithms for this problem. Due to their time efficiency and simplicity, greedy algorithms are of practical importance. We introduce a new analysis showing that when no negative strings contain positive strings, a greedy algorithm achieves $O(n^{4/3})$ and $O(n)$ if the number of negative examples is further bounded by some constant.

Correspondence to: T. Jiang, Department of Computer Science, McMaster University, Hamilton, Ont., L8S 4K1, Canada. Email: jiang@maccs.mcmaster.ca.

[1]Supported in part by NSERC Operating Grant OGP0046613.
[2]Supported in part by the NSERC Operating Grant OGP0046505.

SSDI 0304-3975(93)E0201-E

1. Introduction

Given a finite set of strings, the *shortest common superstring* problem is to find a shortest possible string s such that every string in the set is a substring of s. In a more general setting, given a finite set of *positive* strings and a finite set of *negative* strings, the *shortest consistent superstring* problem is to find the shortest possible string s such that every positive string is a substring of s and no negative string is a substring of s.

A solution of the latter problem implies a solution to the former problem. Both problems have applications in data compression practice and DNA sequencing procedures [8, 9, 12, 13]. The former problem is well-known. The latter problem occurs when merging certain strings is prohibited. For example, in a shot-gun DNA sequencing procedure, certain combinations of fragments make no biological sense and are thus excluded. This problem is especially important in the recently developed DNA sequencing by hybridization (SBH) technique [2, 10]. In an SBH procedure, a biochemist first tests the membership of a large number of oligonucleotide probes (i.e., short strings of nucleotides) in the target sequence and then tries to infer the target sequence. The second step is essentially to construct a superstring consistent with the observed membership of the oligonucleotides.

The latter problem can also be formulated as a learning problem: given a set of positive examples, as substrings of a string to be learned, and a set of negative examples, as strings that are not substrings of the string to be learned, finding a short consistent superstring implies efficient learning in Valiant's PAC learning model [6, 9, 16].

Both problems are NP-hard [3, 4]. Only recently, the open problem of how to approximate a shortest common superstring with constant factors has been partially solved in [6, 9] and completely solved in [1]. [1] shows several greedy algorithms produce a superstring of length $O(n)$ for a given set of strings, where n is the optimal length. But it is not known how such algorithms would perform in the presence of negative strings. The Group-Merge algorithm in [6, 9] actually works even in the presence of negative strings. But that algorithm only achieves an $O(n \log n)$ approximation.

It turns out that negative strings are very hard to deal with. We can show that none of the above algorithms achieve linear approximation. In particular, we will give an $\Omega(n^{1.5})$ lower bound for the greedy algorithms. An $\Omega(n \log n)$ lower bound for Group-Merge is shown in [6]. Thus we have to develop new algorithms and new proofs to solve the question. Remember a linear approximation algorithm for the shortest consistent superstring problem is also a linear approximation algorithm for the shortest common superstring problem.

We give polynomial-time approximation algorithms that produce a consistent superstring of length $O(n)$ for several important special cases, viz., (a) when no negative strings contains positive strings and (b) when there are only a constant number of negative strings. This implies a $\log n$ multiplicative factor improvement on the sample complexity for string learning [6], when the negative examples satisfy one

of these special properties. Case (a) is interesting in practice since it corresponds to the situation when restrictions are imposed on the merging of a pair of input strings. Also the assumption seems to hold in an SBH procedure since the strings involved (i.e., oligonucleotides) usually have roughly the same length [10]. Case (b) arises in a shot-gun DNA sequencing procedure where usually only a limited number of combinations are disallowed. The algorithms all rely on the Hungarian algorithm to find optimal cycle covers on some weighted graphs derived from the input strings in their first stages.

Our other main goal is to study greedy algorithms, because of their practical importance. Although we have obtained a polynomial-time algorithm that linearly approximates the shortest consistent superstring for several important special cases, the polynomial power is too large to be practical. For example, it is not uncommon that we are given 10^6 strings of 100 characters to compress. Simple greedy algorithms running in $O(ml_{\max} \log m)$ time [14, 15], where m and $l_{\max}$ are the number and maximum length of input strings, would require about 10^8 steps. On a supercomputer this, say, takes one second. However Group-Merge would take a million seconds on the same computer, and our new polynomial algorithms would take billions of seconds to finish the task. Remember also, due to their simplicity and efficiency, greedy algorithms are actually implemented and are used routinely by computers or human hands in biochemistry labs and elsewhere. It is thus of great interests to study the performance of greedy algorithms in the presence of negative strings. In Section 4, we will introduce a new method for analyzing greedy algorithms and show that when no negative string contains positive strings as substrings, a greedy algorithm achieves $O(n^{4/3})$ approximation and, moreover, when there are only a constant number of such negative strings it achieves $O(n)$ approximation. We also conjecture that the greedy algorithm actually achieves $O(n)$ when no negative string contains positive strings and hope that some of the analysis techniques developed in this paper will be useful in proving such a linear bound.

2. Preliminaries

Let $P = \{s_1, \ldots, s_m\}$ be a set of *positive* strings and $N = \{t_1, \ldots, t_k\}$ a set of *negative* strings, over some alphabet Σ. Without loss of generality, we assume that sets P and N are "substring free", i.e., no string s_i (or t_i) is contained in any other string s_j (or t_j, respectively). Moreover, we assume that no negative string t_i is a substring of any positive string s_j. A *consistent superstring* for (P, N) is a string s such that each s_i is a substring of s and no t_i is a substring of s. In this paper, we will use n and $OPT(P, N)$ interchangeably for the length of a *shortest* consistent superstring for (P, N). Our goal is to find a consistent superstring for (P, N) whose length is as close to $OPT(P, N)$ as possible.

Since it is NP-hard to decide if (P, N) has a consistent superstring if we require a superstring s to be a string over the same alphabet Σ [7], in this paper we will allow

s to be a string over $\Sigma \cup \{\#\}$, where $\# \notin \Sigma$ is a *delimiter* symbol, so that a trivial consistent superstring (i.e., $s_1 \# s_2 \# \cdots \# s_m\}$ always exists. Note that this assumption is actually consistent with some practice. E.g. when compressing a set of strings into a single superstring, we can always introduce new delimiters if necessary.

Most of the following definitions are introduced in [1]. For two distinct strings s and t, let v be the longest string such that $s = uv$ and $t = vw$. We call $|v|$ the (amount of) *overlap* between s and t, and denote it as $ov(s, t)$. Furthermore, u is called the *prefix* of s with respect to t, and is denoted $pref(s, t)$. We call $|pref(s, t)| = |u|$ the *distance* from s to t, and denote it as $d(s, t)$. So, the string $uvw = pref(s, t)t$, of length $d(s, t) + |t| = |s| + |t| - ov(s, t)$ is the shortest superstring of s and t in which s appears (strictly) before t, and is also called the *merge* of s and t and denoted $m(s, t)$. It is useful to extend the above definitions to also include the case $s = t$. The *self-overlap* of a string s, denoted $ov(s, s)$, is the length of the longest string v such that $s = uv = vw$ for some *nonempty* strings u and w. The extension of the other definitions is straightforward. For $s_i, s_j \in P$, we will abbreviate $pref(s_i, s_j)$ to simply $pref(i, j)$.

The *factor* of a string s, denoted $factor(s)$ is the shortest string u such that $s = u^i v$ for some positive integer i and prefix v of u (v may be null). The *period* of s, denoted by $period(s)$, is $|factor(s)|$. A string s is said to be *i-periodic* if $i \leqslant |s|/period(s) < i + 1$. A string is *fully periodic* if it is at least 4-periodic. A string s is *prefix-periodic* (or *suffix-periodic*) if s is not fully periodic and s has a fully periodic prefix (or suffix, respectively) of length at least $3|s|/4$. (The reason for choosing the specific numbers 4 and 3/4 here can be seen from the proof of Theorem 2.) Call a string *periodic* if it is either fully periodic or prefix-periodic or suffix-periodic. Suppose s is a prefix-periodic string and $s = uv$, where u is the longest fully periodic prefix of s. Then u is called the *periodic prefix* of s and v is the *non-periodic suffix* of s. Similarly, if s is a suffix-periodic string and $s = uv$, where v is the longest periodic suffix of s, then v is called the *periodic suffix* of s and u is the *non-periodic prefix* of s.

For example, $s_0 = \text{abababababa}$ is a fully periodic string with $factor(s_0) = \text{ab}$ and $period(s_0) = 2$, $s_1 = \text{abababababacc}$ is a prefix-periodic string with periodic prefix $u_1 = \text{abababababa}$ and non-periodic suffix $v_1 = \text{cc}$, and $s_2 = \text{bbbbabababab}$ is a suffix-periodic string with non-periodic prefix $u_2 = \text{bbb}$ and periodic suffix $v_2 = \text{babababab}$.

Two strings are *equivalent* if they are cyclic shifts of each other. Two fully periodic strings are *compatible* if they have equivalent factors. Two prefix- (or suffix-) periodic strings are *compatible* if one of their periodic prefixes (suffixes, resp.) is a suffix (prefix, resp.) of the other, and one of their non-periodic suffixes (prefixes, resp.) is a prefix (suffix, resp.) of the other. Informally speaking, two periodic strings have a "large" overlap if and only if they are compatible. Note that compatibility is an equivalence relation.

For example, string $s_1 = \text{abababababc}$ is compatible with $s_2 = \text{babababababcd}$, but not with $s_3 = \text{abababababac}$ and $s_4 = \text{bababababdd}$.

Given a list of strings $s_{i_1}, s_{i_2}, \ldots, s_{i_r}$, we define the superstring $s = \langle s_{i_1}, \ldots, s_{i_r} \rangle$ to be the string $pref(i_1, i_2)pref(i_2, i_3) \cdots pref(i_{r-1}, i_r)s_{i_r}$. That is, s is obtained by maximally overlapping $s_{i_1}, s_{i_2}, \ldots, s_{i_r}$ *in order*. Define $first(s) = s_{i_1}$ and $last(s) = s_{i_r}$. Note, for

two strings s and t obtained by merging strings in P, $ov(s,t)$ in fact equals $ov(last(s), first(t))$, and as a result, the merge of s and t is $\langle first(s), \ldots, last(s), first(t), \ldots, last(t)\rangle$. Observe that since P is substring-free, a shortest common superstring for P must be $\langle s_{i_1}, \ldots, s_{i_m}\rangle$ for some permutation $\langle i_1, \ldots, i_m\rangle$. However, this is not true when there are negative strings, i.e., a shortest consistent superstring for (P, N) may not be $\langle s_{i_1}, \ldots, s_{i_m}\rangle$ for any permutation $\langle i_1, \ldots, i_m\rangle$, since adjacent strings in the superstring may or may not be maximally overlapped.

For any weighted digraph G, a *cycle cover* (or *path cover*) of G is a set of vertex-disjoint cycles (or paths, resp.) covering all nodes of G. (A cycle cover is called an *assignment* in [1], due to the fact that its linear programming form is the same as the well-known assignment problem in operations research.) The cover is said to be optimal if it has the smallest weight. Denote the weight of an optimal cycle cover of graph G as $CYC(G)$. We will consider cycle covers on a weighted complete digraph G_P derived from the positive strings. Digraph $G_P=(V, E, d)$ has m vertices $V=\{1, \ldots, m\}$, and m^2 edges $E=\{(i, j)\colon 1\leqslant i, j\leqslant m\}$. Here we take as weight function the distance $d(,)$: edge (i, j) has weight $w(i, j)=d(s_i, s_j)$, to obtain the *distance graph*. The string represented by a path $i_1, \ldots, i_r$ is $\langle s_{i_1}, \ldots, s_{i_r}\rangle$. As observed in [1], we have

$$CYC(G_P)\leqslant OPT((P, \emptyset)).$$

Denote by $w(c)$ the *weight* of a cycle c. For convenience, define the length of a cycle c, denoted $l(c)$, to be its weight and the length of a path $p=i_1, \ldots, i_r$, denoted $l(p)$, to be $|\langle s_{i_1}, \ldots, s_{i_r}\rangle| = d(s_{i_1}, s_{i_2}) + \cdots + d(s_{i_{r-1}}, s_{i_r}) + |s_{i_r}|$.

Throughout this paper, by a path (or cycle) we mean a simple path (or cycle), unless otherwise specified.

3. Linear approximation: two special cases

In this section we present polynomial time algorithms which produce a consistent superstring of length $O(n)$, where n is the optimal length, for two special cases: (a) when no positive string is a substring of any negative strings; and (b) when there are only constant number of negative strings.

The best previous algorithm for shortest consistent superstrings is Group-Merge introduced in [6, 9], which achieves $\theta(n \log n)$ approximation [6]. The lower bound in fact holds for $N=\emptyset$. Thus Group-Merge does not work well for the special cases that we are interested in. Although we suspect that greedy algorithms may achieve linear approximation in these special cases, so far we can only prove an upper bound $O(n^{4/3})$ (to be presented in next section). Thus we must search for new algorithms. Our departure point is the algorithm Concat-Cycles which is used to find a common superstring of length $O(n)$ in [1], when there are no negative strings. The essence of Concat-Cycles is the Hungarian algorithm [11] which can find an optimal cycle cover for any given weighted digraph. But, [1] did not need the full power of the Hungarian algorithm. We will fully utilize its power.

3.1. *When no negative strings contain positive ones*

In this subsection, we show that when no negative strings contain positive strings as substrings, an algorithm can achieve linear approximation. This special case is natural, it corresponds to the restrictions that we impose on the merges of input strings. In some practice, we may want to forbid some "bad merges" to happen.

As mentioned above, our algorithm works in a way similar to Concat-Cycles [1] and uses the Hungarian algorithm to find an optimal cycle cover on the distance graph derived from the input strings. It is shown in [1] that if we have an optimal cycle cover of the distance graph, then opening each cycle into a path arbitrarily and simply concatenating the strings associated with the paths yields a superstring of length $O(n)$.

We informally describe the construction of our algorithm. Let $P=\{s_1, \ldots, s_m\}$ and $N=\{t_1, \ldots, t_k\}$. First construct the distance graph G_P for the positive strings as defined in Section 2, except that for each pair i, j such that $m(s_i, s_j)$ contains a negative string we remove the edge (i, j) from G_P. Note that, since no negative strings contain any positive strings, each path in the new G_P corresponds to a string consistent with N.

Several problems need to be solved: (i) G_P may not have a cycle cover; (ii) even if G_P has a cycle cover, it is not clear if $CYC(G_P)=O(n)$, which is essential to achieving an $O(n)$ bound on the length of superstring produced.

(i) is easy to solve. We can just add a sufficient number of the *delimiter nodes* (called # nodes) to G_P. Each such node represents a delimiter #. We set the weights as follows: $w(\#, \#)=0$ and for each node, i $w(i, \#)=|s_i|$ and $w(\#, i)=1$. Call the resulting graph $G_{P\#}$. Clearly $G_{P\#}$ always has a cycle cover. Note that the use of delimiter nodes is consistent with our definition of a superstring. Also observe that $CYC(G_{P\#}) \leqslant CYC(G_P)$ if the latter exists, as we can always let the delimiter nodes form a cycle with zero weight.

But it is still not obvious that $CYC(G_{P\#})=O(n)$. The reason is that in a shortest consistent superstring, two adjacent strings may or may not be maximally overlapped and maximally overlapping two strings sometimes prevents better arrangement, because of the presence of negative strings. So (ii) is resolved by considering a special form of consistent superstrings. First observe that for any pair of strings s and t, if s is not suffix-periodic and t is not prefix-periodic, then there is at most one way of overlapping s and t with s in front to achieve a large amount of overlap (i.e., $\geqslant 3\max\{|s|, |t|\}/4$). Thus, if the overlap between s and t is large, then the overlap must equal $ov(s, t)$. Now define a *normal* superstring for (P, N) to be a superstring of the form: $u_1 \# u_2 \# \cdots \# u_r$, where each u_i is either s_i or $m(s_i, s_j)$ for some $i \neq j$. Denote the length of a shortest normal consistent superstring for (P, N) by $OPT1(P, N)$.

Lemma 1. *If P does not contain any periodic strings, then $OPT1(P, N)=O(n)$.*

Proof. Let s be a shortest consistent superstring for (P, N). Order strings $s_1, \ldots, s_m$ according to their first occurrences in s. Suppose the sequence is $s_{i_1}, \ldots, s_{i_m}$. Cut the sequence into segments *maximally* from left to right such that each segment satisfies

(i) it contains a single string, or (ii) the first and last string in it overlap by at least $\frac{3}{4}$ of their maximum length. Let $(a_1, b_1), \ldots, (a_r, b_r)$ be the pairs of the first and last strings of the segments. For each pair (a_j, b_j) with $a_j \neq b_j$, since, a_j, b_j are non-periodic and their overlap in s is at least $3 \max\{|a_j|, |b_j|\}/4$, they overlap by precisely $ov(a_j, b_j)$ in s. Hence the string $m(a_j, b_j)$ is a consistent superstring covering all the strings in segment j. Let $u_j = m(a_j, b_j)$ if $a_j \neq b_j$ or a_j otherwise. Then $u = u_1 \# \cdots \# u_r$ is a consistent superstring for (P, N). Note that $\sum_{j=1}^{r} |u_j| \leqslant \sum_{j=1}^{r} |a_j| + |b_j|$. The following calculation shows that $\sum_{j=1}^{r} |a_j| < 8|s| = 8n$ and $\sum_{j=1}^{r} |b_j| < 8n$.

Denote by a_j^R the reverse of string a_j. Since a_j and a_{j+1} overlap by less than $3 \max\{|a_j|, |a_{j+1}|\}/4$ in s,

$$d_s(a_j, a_{j+1}) + d_s(a_{j+1}^R, a_j^R) \geqslant \max\{|a_j|, |a_{j+1}|\}/4$$

where $d_s(a_j, a_{j+1})$ is the distance from a_j to a_{j+1} in the superstring s, and similarly, $d_s(a_{j+1}^R, a_j^R)$ is the distance from a_{j+1}^R to a_j^R in the superstring s^R. Hence,

$$d_s(a_j, a_{j+1}) + d_s(a_{j+1}^R, a_j^R) \geqslant |a_j|/4.$$

Since

$$|s| \geqslant |a_r| + \sum_{j=1}^{r-1} d_s(a_j, a_{j+1})$$

$$|s^R| \geqslant |a_1| + \sum_{j=r}^{2} d_s(a_j^R, a_{j-1}^R),$$

we have

$$2|s \geqslant |a_1| + |a_r| + \sum_{j=1}^{r-1} d_s(a_j, a_{j+1}) + d_s(a_{j+1}^R, a_j^R)$$

$$\geqslant |a_1| + |a_r| + \sum_{j=1}^{r-1} |a_j|/4.$$

Thus, $|s| > \sum_{j=1}^{r} |a_j|/8$. Similarly we can prove that $|s| > \sum_{j=1}^{r} |b_j|/8$. □

Observe that in the proof of above lemma, for each segment $j = a_j, \ldots, b_j$, $\langle a_j, \ldots, b_j \rangle = m(a_j, b_j)$, since P is substring-free and the strings in the segment are overlapped by a large amount. Thus the constructed consistent superstring u in fact corresponds to a Hamiltonian path on $G_{P\#}$ and the lemma implies that if P does not contain any periodic strings, then

$$CYC(G_{P\#}) \leqslant OPT1(P, N) = O(n)$$

Although we can actually show that the above result holds for any set P of strings, we do not need the stronger result here since we will process the periodic strings separately anyway, for some other reason.

Before we formally present our linear approximation algorithm, we need to describe a simple greedy algorithm Greedy 1, which is a straightforward extension of the algorithm Greedy discussed in [1, 14, 15].

Algorithm Greedy 1

1. Choose two (different) strings s and t from P such that $m(s,t)$ does not contain any string in N and $ov(s,t)$ is maximized. Remove s and t from P and replace them with the merged string $m(s,t,)$. Repeat Step 1. If such s and t could not be found, go to Step 2.
2. Concatenate the strings in P, inserting delimiters # if necessary.

Our approximation algorithm combines Greedy 1 and the Hungarian algorithm:

1. Put the fully periodic strings in P into set X_1, the prefix-periodic strings into set X_2, the suffix-periodic strings into set X_3, and other strings into set Y.
2. Divide X_1, X_2, and X_3 further into groups of compatible strings. Run Greedy 1 on each group separately.
3. Construct the graph $G_{Y\#}$ as described above. Find an optimal cycle cover of $G_{Y\#}$. Open each cycle into a path and thus a string.
4. Concatenate the strings obtained in steps 2 and 3, inserting #'s if necessary.

Theorem 2. *Given (P,N), where no string in N contains a string in P, the above algorithm produces a consistent superstring for (P,N) of length $O(n)$.*

Proof. (Sketch) We know from the above discussion that the optimal cycle cover found in step 3 has weight $CYC(G_{Y\#}) = O(OPT1(Y,N)) = O(OPT(Y,N)) = O(n)$. Since the strings in Y are non-periodic, it is easy to show that their merges are at most 4-periodic. The strings that are at most 4-periodic do not have large self-overlap. More precisely, $ov(s,s) < 4|s|/5$ for any s that is at most 5-periodic. Thus opening a cycle into a path can at most increase its length by a factor of 5. This shows the strings obtained in Step 3 have a total length at most 5 $CYC(G_{Y\#}) = O(n)$.

Now we consider the strings produced in Step 2. Let $U_1, \ldots, U_r$ be the compatible groups for X_2. (The proof for X_1 and X_3 are similar.) It follows from Lemma 9 in [1] that for any two fully periodic strings x and y, if x and y are incompatible, then $ov(x,y) < period(x) + period(y)$. By our definition of periodicity, for any $u_i \in U_i$, $u_j \in U_j$, $i \neq j$, $ov(u_i,u_j) < (|u_i| + |u_j|)/4 + \max\{|u_i|, |u_j|\}/4 < 3\max\{|u_i|, |u_j|\}/4$. Thus, informally speaking, strings belonging to different groups do not have much overlap with each other. It can be shown by a calculation as in the proof of Lemma 1 that we can afford losing such "small overlaps" in constructing an $O(OPT(X_2,N))$ long consistent superstring for (X_2,N), since replacing each such overlap with a plain concatenation in a shortest consistent superstring for (X_2,N) will at most increase its length by a factor of 8. Hence we have the following lemma:

Lemma 3. $\sum_{i=1}^{r} OPT(U_1,N) = O(OPT(X_2,N)) = O(n)$.

To complete the proof, it suffices to prove that Greedy 1 produces a consistent superstring of length $O(OPT(U_i, N))$ for each group U_i. A key observation in this proof is that because the strings in U_i are all compatible with each other, the large overlaps are unique in the following sense: for any $s, t \in U_i$, if $m(s,t)$ does not contain any negative examples, then $ov(s,t)$ must occur in the construction of a shortest consistent superstring. Thus, Greedy 1 can actually identify all the *correct* (i.e., used in the construction of a shortest consistent superstring) large overlaps and perform the corresponding merges. Greedy 1 will ignore all the small overlaps (including the correct ones) and replaces them with concatenation. But this is fine as observed before. □

Remark. Since the Hungarian algorithm runs in $O(m^3)$ time on a graph of m nodes, our algorithm has time complexity $O(m^3 l_{\max})$, where $l_{\max}$ is the maximum length of the input strings.

3.2. With constant number of negative strings

In this section, we consider the case $|N| \leqslant c$, for some constant c. However here we allow any kind of negative strings. We present a linear approximation algorithm for this special case. A sketch of the construction of our algorithm is given below.

Again, given input (P, N) with $|N| \leqslant c$, we remove the periodic strings from P and process them separately to obtain a consistent superstring of length $O(n)$ for these periodic strings, as shown in the previous subsection. So from now on assume that P does not contain any periodic strings.

Let G_P be the distance graph for P as defined in Section 2. Again obtain the graph $G_{P\#}$ by adding a sufficient number of delimiter nodes to G_P. Now we have to find an optimal cycle cover of $G_{P\#}$ that is consistent with N, i.e., each cycle should not contain a path whose associated string violates the negative strings in N. Let $CYC(G_{P\#}, N)$ denote the weight of an optimal consistent cycle cover of graph $G_{P\#}$. By the proof of Lemma 1, we have

$$CYC(G_{P\#}, N) \leqslant OPT1(P, N) = O(n).$$

It is not easy to find an optimal consistent cycle cover directly. So our construction has to utilize the fact that $|N| \leqslant c$. Our main idea is to make many (but polynomial number) copies of $G_{P\#}$; each is slightly modified (some edges deleted) according to the negative strings. The graphs are constructed so that the inconsistent cycle covers are prevented. In other words, in these graphs, every cycle cover is consistent with N. We also want these graphs to give all possible consistent cycle covers of $G_{P\#}$ collectively. Thus by running the Hungarian algorithm on each of them we can find an optimal consistent cycle cover of $G_{P\#}$.

We describe how to construct these graphs. Consider a negative string t. Observe that there may be many paths in $G_{P\#}$ violating t. We have to prevent them all. Let

$i_1, \ldots, i_r$ be the path (not necessarily simple) with maximum number of nodes in $G_{P\#}$ such that $\langle s_{i_1}, \ldots, s_{i_r} \rangle$ is contained in t. Observe that the path is unique. Let P_1 include the strings in P such that merging any string in P_1 to the left of $\langle s_{i_1}, \ldots, s_{i_r} \rangle$ would cover a prefix of t and, let P_2 include the strings in P such that merging any string in P_2 to the right of $\langle s_{i_1}, \ldots, s_{i_r} \rangle$ would cover a suffix of t. For convenience, let s_{i_0} denote a string in P_1 and $s_{i_{r+1}}$ denote a string in P_2 generically. The following lemma essentially says that in each consistent cycle cover of $G_{P\#}$, the path $i_0, i_1, \ldots, i_r, i_{r+1}$ has to be completely broken.

Lemma 4. *Each consistent cycle cover of $G_{P\#}$ can be transformed into one without increasing the weight such that there is an index $0 \leqslant j \leqslant r$ such that none of the edges of the form (i_a, i_b), where $a \leqslant j < b$ and $m(s_{i_a}, s_{i_b}) = \langle s_{i_a}, \ldots, s_{i_b} \rangle$, are used in the cover.*

Proof. Let C be a consistent cycle cover of $G_{P\#}$. We transform C and find the index j as follows. If C has no edge of the form (i_0, i_a), where $a > 0$ and $m(s_{i_0}, s_{i_a}) = \langle s_{i_0}, \ldots, s_{i_a} \rangle$, then we can simply choose $j = 0$. Otherwise let (i_0, i_a) be such an edge. We can "transfer" all the vertices $i_1, \ldots, i_{a-1}$ to this edge without increasing the weight. Then we check if the edge from i_a in C is of the form (i_a, i_b), where $b > a$ and $m(s_{i_a}, s_{i_b}) = \langle s_{i_a}, \ldots, s_{i_b} \rangle$. If not, then we let $j = a$. Otherwise we repeat the above procedure. The whole process must terminate before we reach index i_{r+1} since C is consistent with t. □

Thus, for the negative string t, we construct $r + 1 \leqslant m = |P|$ graphs $G_t^0, G_t^1, \ldots, G_t^r$ from $G_{P\#}$ by deleting some edges. We make sure that in each G_t^j, all the edges (i_a, i_b) satisfying the condition in the above lemma are broken. Thus each cycle cover of G_t^j is consistent with the string t. By the lemma, there exists some j such that an optimal consistent cycle cover of G_t^j is also an optimal consistent cycle cover of $G_{P\#}$. Then starting again from each G_t^j, we repeat the above procedure for another negative string, constructing more graphs. This eventually gives us at most m^c graphs. One of such graphs must contain an optimal consistent cycle cover of $G_{P\#}$.

So the last step of our algorithm is to run the Hungarian algorithm and obtain an optimal cycle cover for each of these graphs, and choose a cycle cover with the minimum weight. By the above discussion, the chosen cycle cover is an optimal consistent cycle cover of $G_{P\#}$. Then we simply open the cycles into paths and concatenate the corresponding strings. Since the strings are non-periodic, this results in a superstring of length at most $5CYC(G_{P\#}, N) \leqslant 5n$.

Theorem 5. *Given (P, N) with $|N| \leqslant c$ for some constant c, the above algorithm outputs a consistent superstring of length $O(n)$.*

Remark. It is possible to give an alternative proof of the above theorem using overlap properties of negative strings when opening cycles, but this analysis would give a higher approximation factor.

4. Greedy solutions

Our main objective is to analyze the greedy-style algorithms in the presence of negative strings. This, we believe, will also better our understanding about the original shortest common superstring problem. Because of their simplicity, time-efficiency, and appeal to common sense, greedy algorithms are routinely used in computer programs and by human hands. For example, greedy algorithms usually run in $O(ml_{\max} \log m)$ time [14, 15], where m and $l_{\max}$ are the number and maximum length of input strings, whereas our new algorithms and Group-Merge would require at least $O(m^3 l_{\max})$ time. In practice, it is possible to have a large number of input strings, and in such cases, it is infeasible to use algorithms with time complexity $\Omega(m^2 l_{\max})$. This makes greedy algorithms the only practical algorithms known so far.

Although greedy algorithms can achieve $O(n)$ approximation when there are no negative strings, our next theorem shows that generally they do not work well when negative strings are present. For simplicity, we just prove a lower bound for the algorithm Greedy 1 described in the previous section.

Theorem 6. *For some input* (P, N), *Greedy* 1 *produces a superstring of length* $\Omega(n^{1.5})$.

Proof. We will construct two sets of strings P, N to force Greedy 1 to output a superstring of length $\theta(n^{1.5})$. The true shortest superstring in our mind is: $b^n a^k \# a^k b^n$ where $k = \theta(\sqrt{n})$. P contains $a \# a$, and the pairs of positive strings: $b^{n+1-i(i+1)/2} a^i$, $a^i b^{n+1-i(i+1)/2}$, $1 \leqslant i \leqslant \sqrt{n}$. If $b^i a^j$, $a^j b^i$ is in P, then N contains $a^{j+1} b^i a^j$ and $a^j b^i a^{j+1}$.

Thus the first pair of strings in P will merge (wrongly) to $ab^n a$, negative strings $a^2 b^n a$ and $ab^n a^2$ will prevent further merge to $ab^n a$. Then the second pair in P will merge to $a^2 b^{n-2} a^2$, and so on.

Thus we will end up with $\sqrt{n}$ strings of form $a^i b^j a^i$ with total length $\Omega(n^{1.5})$. Because of negative strings in N, they must be concatenated to a final string of length $\Omega(n^{1.5})$. □

If we inspect the construction in the above proof carefully, we observe that some positive strings are substrings of negative strings. Such positive strings trick Greedy 1 into bad traps. If we forbid such things to happen, can a greedy algorithm do better? The answer turns out to be positive. Our result will show that negative strings which contain positive strings are essentially responsible for the bad cases. In the following, we present a greedy algorithm which produces a consistent superstring of length $O(n^{4/3})$, when no negative string contains positive strings. (Recall that we have given an $O(n)$ approximation algorithm for this special case in Section 3, with time complexity $O(m^3 l_{\max})$.) The algorithm combines Greedy 1 with another algorithm Mgreedy 1, which is a straightforward extension of the algorithm Mgreedy in [1]. We first describe Mgreedy 1.

Algorithm Mgreedy 1

1. Let (P, N) be the input and T be empty.
2. While P is non-empty, do the following: Choose $s, t \in P$ (not necessarily distinct) such that $m(s, t)$ does not contain any string in N and $ov(s, t)$ is maximized. If $s \neq t$, then remove s and t from P and replace them with the merged string $m(s, t)$. If $s = t$, then just move s from P to T. If such s and t could not be found, move all strings in P to T.
3. Concatenate the strings in T, inserting delimiters # if necessary.

It is not easy to prove a nontrivial upper bound on the performance of Greedy 1, nor is it easy for Mgreedy 1. The trouble maker again is the periodic strings. So we will consider an algorithm which processes the periodic and non-periodic strings separately:

1. Put the fully periodic strings in P into set X_1, the prefix-periodic strings into set X_2, the suffix-periodic strings into set X_3, and other strings into set Y.
2. Divide X_1, X_2, and X_3 further into groups of compatible strings. Run Greedy 1 on each group separately.
3. Run Mgreedy 1 on set Y.
4. Concatenate the strings obtained in Steps 2 and 3, inserting #'s if necessary.

Theorem 7. *Given (P, N), where no string in P is a substring of a string in N, the above algorithm returns a consistent superstring of length $O(n^{4/3})$.*

Proof. By the proof of Theorem 2, the strings produced in Step 2 have total length $O(n)$. So it remains to analyze Step 3. Let s_Y be a shortest consistent superstring for (Y, N). Then clearly $|s_Y| \leqslant n$. Again observe that the strings in Y do not have a long periodic prefix or suffix. Thus, for any two strings s and t there is at most one way of overlapping them to achieve a large amount of overlap (i.e., $\geqslant 3 \max\{|s|, |t|\}/4$).

The proof of the $O(n)$ bound for Mgreedy in [1] essentially uses the fact that Mgreedy actually selects the edges (representing merges) following a *Monge sequence* on the distance graph derived from the given strings. (For a definition of Monge sequences see, e.g., [5].) Thus Mgreedy first finds an optimal cycle cover of the distance graph. However, with the presence of negative strings, a distance graph may or may not have a Monge sequence. (The negative strings forbid some edges.) Thus we have to use a different strategy. Our analysis scheme can be roughly stated as follows. Again consider the distance graph G_Y and view Mgreedy 1 as choosing edges in the graph. When Mgreedy 1 merges two strings s and t, it chooses the edge from $last(s)$ to $first(t)$. Initially, we fix a path cover $\mathscr{C}$ on G_Y such that the total length of the paths in $\mathscr{C}$ is $O(|s_Y|)$. We analyze Mgreedy 1 on Y with respect to the initial cover $\mathscr{C}$. As Mgreedy 1 merges strings, we update the cover by possibly breaking a path into two or joining two paths into one or turning a path into a cycle. The merges performed by Mgreedy 1 are divided into several classes. A merge is *correct* if it chooses an edge in some current path or cycle. Otherwise the merge is *incorrect*. An incorrect merge is a *jump merge* if it breaks two potential correct merges simultaneously. Suppose in

a jump merge Mgreedy 1 chooses an edge (x, y). Let x' be the current successor of x and y' the current predecessor of y, in their respective paths/cycles. That is, the choice of edge (x, y) prevents us from choosing the edges (x, x') and (y', y) in the future. Then the merge is *good* if $m(y', x')$ does not contain any negative string. Otherwise the merge is *bad*. Clearly the type of a merge performed by Mgreedy 1 depends on the initial cover $\mathscr{C}$ and how we update paths and cycles. We will use the following updating rule. Suppose that Mgreedy 1 chooses an edge (x, y).

Case 1: The merge is correct. No change.

Case 2: The merge is a good jump.

Subcase 2.1: x and y are from a same path $a, \ldots, b, x, c, \ldots, d, y, e, \ldots, f$. Split the path into a path and a cycle: $a, \ldots, b, x, y, e, \ldots, f$, and $c, \ldots, d, c$.

Subcase 2.2: x and y are from a same cycle $a, \ldots, b, x, c, \ldots, d, y, e, \ldots, f, a$. Split the cycle into two cycles: $a, \ldots, b, x, y, e, \ldots, f, a$ and $c, \ldots, d, a$.

Subcase 2.3: x and y are from two paths $a, \ldots, b, x, c, \ldots, d$ and $e, \ldots, f, y, g, \ldots, h$. Shift the paths: $a, \ldots, b, x, y, g, \ldots, h, e, \ldots, f, c, \ldots, d$.

Subcase 2.4: x and y are from two cycles $a, \ldots, b, x, a$ and $c, \ldots, d, y, c$. Combine the two cycles into one cycle: $a, \ldots, b, x, y, c, \ldots, d, a$.

Subcase 2.5: x and y are from a path and a cycle $a, \ldots, b, x, c, \ldots, d$ and $e, \ldots, f, y, e$. Open the cycle and insert it into the path: $a, \ldots, b, x, y, e, \ldots, f, c, \ldots, d$.

Case 3: The merge is a bad jump.

Subcase 3.1: x and y are from a same path $a, \ldots, b, x, c, \ldots, d, y, e \ldots, f$. Split the path into two paths: $a, \ldots, b, x, y, e, \ldots, f$ and $c, \ldots, d$.

Subcase 3.2: x and y are from a same cycle $a, \ldots, b, x, c, \ldots, d, y, e, \ldots, f, a$. Split the cycle into a cycle and a path: $a, \ldots, b, x, y, e, \ldots, f, a$ and $c, \ldots, d$.

Subcase 3.3: x and y are from two paths $a, \ldots, b, x, c, \ldots, d$ and $e, \ldots, f, y, g, \ldots, h$. Shift the paths and create three paths: $a, \ldots, b, x, y, g, \ldots, h, c, \ldots, d$, and $e, \ldots, f$.

Subcase 3.4: x and y are from two cycles $a, \ldots, b, x, a$ and $c, \ldots, d, y, c$. Open the cycles and combine them into a path: $a, \ldots, b, x, y, c, \ldots, d$.

Subcase 3.5: x and y are from a path and a cycle $a, \ldots, b, x, c, \ldots, d$ and $e, \ldots, f, y, e$. Split the path into two paths and combine one with the cycle: $a, \ldots, b, x, y, e, \ldots, f$ and $c, \ldots, d$.

Case 4: The merge is incorrect, but not a jump. In this case, x or y must be the head or tail of some path. Suppose x is the tail of some path $a, \ldots, b, x$.

Subcase 4.1: y is from a path $c, \ldots, d, y, e, \ldots, f$. Split the path into two and combine one with the path containing x: $a, \ldots, b, x, y, e, \ldots, f$ and $c, \ldots, d$.

Subcase 4.2: y is from a cycle $c, \ldots, d, y, c$. Open the cycle and combine it with the path containing x: $a, \ldots, b, x, y, c, \ldots, d$.

The following lemma [14, 15] implies that only bad jump merges can increase the total length of paths/cycles.

Lemma 8. *Let x, y, x', y' be strings, not necessarily different, such that $ov(x, y) \geq \max\{ov(x, x'), ov(y', y)\}$. Then, $d(x, y) + d(y', x') \leq d(x, x') + d(y', y)$.*

Our objective is to show that when Mgreedy 1 terminates, the total length of paths/cycles is $O(|s_Y|^{4/3})$. This is achieved by first proving an upper bound $O(|\mathscr{C}|^{3/2})$ on the total number of bad jump merges performed by Mgreedy 1. For this, we actually need the initial path cover $\mathscr{C}$ to satisfy two more conditions: (i) all jump merges performed by Mgreedy 1 are bad with respect to $\mathscr{C}$; (ii) the strings in each initial path must overlap "a lot", i.e., the overlap must be at least 11/12 of their maximum length.

Lemma 9. *There exists a path cover $\mathscr{C}$ such that*: (i) *The total length of $\mathscr{C}$ is $O(|s_Y|)$*; (ii) *All jump merges performed by Mgreedy 1 are bad with respect to $\mathscr{C}$.*

Proof. The superstring s_Y naturally defines a cover $\mathscr{C}_0$ consisting of a single path of length $|s_Y|$. We now consider merges performed by Mgreedy 1 on Y with respect to $\mathscr{C}_0$, but paying attention only to good jump merges. When there is a good jump merge, we rearrange the path/cycles as in above Case 2. This will not increase the total length of paths/cycles. At the end of the process, we open each cycle into a path. Since the strings in Y are non-periodic, the total length will at most be increased by a factor of 6 as discussed in the proof of Theorem 2. Let $\mathscr{C}_1$ denote the resulting path cover. Then if we use $\mathscr{C}_1$ as the initial cover and update the paths/cycles according to the above rule, all jump merges performed by Mgreedy 1 Y will be bad. □

We then refine $\mathscr{C}_1$ to satisfy the second condition: Divide each path into subpaths such that

(i) the first and last strings of each subpath overlaps by at least 22/23 of their maximum length;

(ii) the first strings of two adjacent subpaths overlaps by less than 22/23 of their maximum length. Observe that the strings belonging to a same subpath are of roughly the same length: The longest to shortest ratio is at most 24/22. Hence every pair of strings in a subpath overlap by at least $22/24 = 11/12$ of their maximum length. Let $\mathscr{C}_2$ denote the resulting cover and l_0 denote its total length. It follows from the proof of Lemma 1 that $l_0 = O(|s_Y|)$.

Now we bound the number of bad merges using $\mathscr{C}_2$ as the initial path cover. From now on, by a path, we mean an initial path in $\mathscr{C}_2$. It is not hard to see that since there are no good jump merges, now a correct merge actually chooses an edge that exists in some (initial) path. Thus a bad jump merge breaks two edges, both exist in some (initial) paths. Also note because Mgreedy 1 always chooses a largest overlap, the overlap achieved in a bad jump merge is no less than the two broken "correct" overlaps.

Lemma 10. *No bad merge can involve two strings from a same path.*

Proof. Because strings in a path overlap "a lot", if a bad merge happens within the path, the sandwiched strings would be periodic and thus a contradiction. □

If Mgreedy 1 performs a bad merge $m(s, t)$, we say that string *first*(t) *interferes* with string *last*(s) and call the merge an *interference*. If a string from a path A interferes with a string from a path B, we say that A interferes with B. By above lemma, all interferences are between different paths. Again, in the following by a large overlap we mean one with length at least 3/4 of the maximum length of the two involved strings. The next lemma shows that if path A interferes with path B, then strings in A have large overlaps with strings in B. The lemma should also explain why we want the overlap ratio between the strings in a path to be at least 11/12.

Lemma 11. *Let* a_1, a_2, b_1, b_2 *be four strings such that* $ov(a_1, a_2) \geqslant 11 \max\{|a_1|, |a_2|\}/12$ *and* $ov(b_1, b_2) \geqslant 11 \max\{|b_1|, |b_2|\}/12$. *If* $ov(a_2, b_1) \geqslant \max\{ov(a_1, a_2), ov(b_1, b_2)\}$, *then* $ov(a_1, b_2) \geqslant 33 \max\{|a_1|, |b_2|\}/4$.

Proof. It is easy to see that

$$ov(a_1, b_2) \geqslant ov(a_2, b_1) - d(b_1, b_2) - d(a_2^R, a_1^R).$$

Now clearly

$$ov(a_2, b_1) \geqslant 11 \max\{|a_1|, |a_2|, |b_1|, |b_2|\}/12$$

$$d(b_1, b_2) = |b_1| - ov(b_1, b_2) \leqslant |b_1|/12$$

$$d(a_2^R, a_2^R) = |a_2| - ov(a_1, a_2) \leqslant |a_2|/12.$$

Thus,

$$ov(a_1, b_2) \geqslant 9 \max\{|a_1|, |a_2|, |b_1|, |b_2|\}/12 \geqslant 3 \max\{|a_1|, |b_2|\}/4. \quad \square$$

Lemma 12. *A path can interfere with another path at most once.*

Proof. Without loss of generality, assume that a_1 and a_2 from path A interfere with b_f and b_r belonging to path B, respectively, where b_f appears before b_r in path B. This assumption implies that a_1 appears before a_2 in path A, since otherwise the string b_r would be periodic, because of the large amount of overlaps involved in the interferences.

By lemma 11, the fact that a_1 interferes with b_f implies that the predecessor of a_1 in A has a large overlap with the successor of b_f in B, but this overlap would violate a negative string. (Note that a_1 is not the head of path A since otherwise there will be no interference.) However since a_2 interferes with b_r, merging the predecessor of a_1 with a_2 would also violate this same negative string. This is a contradiction since the

predecessor of a_1 overlaps a lot with a_2 in path A and there is only one way of overlapping the two strings by a large amount. □

Lemma 13. *It cannot happen that path A interferes with path B, and path B interferes with path A.*

Proof. Without loss of generality, assume that a_f of path A interferes with b_1 of path B and b_2 of path B interferes with a_r of path A. This implies that b_2 is behind b_1 in B since otherwise a_f and a_r would be periodic.

The fact a_f interferes with b_1 implies that the predecessor of a_f has a large overlap with the successor of b_1 in B, and thus with b_2, but such an overlap would violate some negative string. Hence the fact that b_2 interferes with a_r implies that the subpath of A before a_r contains the same negative string, a contradiction. □

We now do a preliminary estimate on the total length increase caused by the interferences. Let $\mathscr{C}_2$ be $\{p_1, \ldots, p_m\}$, where $l(|p_1|) \geqslant \cdots \geqslant l(|p_m|)$. Every time there is an interference, we charge a cost equal to the minimum length of the two involved paths. Obviously the worst scenario is that the interferences involve as many lower-indexed paths as possible and all involved paths have the same length. By the above lemmas, a path p_i can get charged at most $i-1$ times. Clearly the total number of interferences is at most $|Y| \leqslant |s_Y|$. Thus the worst case happens when $m=(2|s_Y|)^{1/2}$ (roughly), p_i interferes $i-1$ times for each $i=1,2,\ldots,m$, and the paths have length $l_0/m=O(|s_Y|/m)=O(|s_Y|^{1/2})$. This already gives a non-trivial upper bound $O(|s_Y|^{3/2})$ on the total length increase caused by the interferences. We can improve this bound to $O(|s_Y|^{4/3})$ by studying the structure of interferences in more detail.

Lemma 14. *No two paths can interfere with three common paths. That is, there cannot be five paths A, B, C, D, E such that both A and B interfere with C, D, E.*

Proof. Suppose A and B interfere with C, D, E. Out of C, D, E, there must be two paths, say C, D, and out of A, B there must be one path, say A, such that a_f interferes with c_r, and a_r interferes with d_r, where a_f appears before a_r in A, and c_r, d_r appear after c_f, d_f, which are interfered by strings from B, resp. in paths C, D. It is sufficient to consider the following two cases.

Case 1: b_f interferes with c_f and b_r interferes with d_f. The fact that b_r interferes with d_f implies that b_f has large overlap with the strings in path D behind d_f, hence d_r, hence a_f, hence a_r, and this overlap between b_f to a_r violates some negative string. Since b_f has no long periodic suffix, its merge with a_r by a large amount is unique. But b_f interferes with c_f and a_r interferes with c_r. This implies that the merge between c_r and a_r must violate the above negative string, because from c_f to c_r in C, all the merges do not violate this negative string and no negative string contains a positive string a positive string as a substring.

Case 2: b_f interferes with d_f and b_r interferes with c_f. The fact the b_r interferes with c_f implies that c_r, hence b_f has a large overlap with a_f, but a merge of them is prevented by a negative string. However the fact d_r has a large overlap with a_f implies that d_r also has a large overlap with a_r. Since b_f interferes with d_f and a_f has just a unique way of overlapping with a_r by a large amount, we conclude that a_f also has a large overlap with b_f.

But we know that a merge between b_f and a_r violates a negative string. This negative string must also be violated by a merge of b_f to a_f because no negative string contains a positive string. But from d_f to d_r, all the merges do not violate negative strings. Therefore it must be the case that either the merge of d_r and a_f or the merge of d_f and b_f violates the negative string.

We have shown that both of these cases are not possible. The lemma follows. □

We next prove a simple combinatorial lemma.

Lemma 15. *Let S be a set of size k. Suppose $S_1, \ldots, S_k$ are k subsets of S such that $S_i \cap S_j \leqslant c$, for any $i \neq j$ and some constant c. Then $\sum_{i=1}^{k} |S_i| \leqslant O(k^{3/2})$.*

Proof. It suffices to show that at most $\sqrt{k}-1$ of the subsets $S_1, \ldots, S_k$ can have $(c+1)\sqrt{k}$ or more elements. Suppose for the contradiction that $S_1, \ldots, S_{\sqrt{k}}$ have $(c+1)\sqrt{k}$ or more elements. Then $|S_1| + \cdots + |S_{\sqrt{k}}| > (c+1)k$. However, this is impossible since these subsets can share at most $\binom{\sqrt{k}}{2} c < kc$ elements totally. □

The above upper bound is actually tight since one can construct the subsets such that $|S_i \cap S_j| \leqslant 1$ for any $i \neq j$ and $\sum_{i=1}^{k} |S_i| \geqslant k^{3/2}$.

Recall that m is the number of paths in $\mathscr{C}_2$. Obviously $m \leqslant |s_Y|$. It follows from Lemmas 14 and 15 that the total number of interferences is $O(m^{3/2})$.

Lemma 16. *The total length increase caused by interferences is $O(|s_Y|^{4/3})$.*

Proof. We estimate the cost of the interferences as before. Let $\mathscr{C}_2$ be $\{p_1, \ldots, p_m\}$, where $l(p_1) \geqslant \cdots \geqslant l(p_m)$. Again, for each interferences we charge a cost equal to the minimum length of the two involved paths to the shorter path. The worst scenario is still that the interferences involve as many lower-indexed paths as possible and all involved paths have the same length. Now, according to Lemma 14, the worst case happens when $m = n^{2/3}$, there are totally $|s_Y|$ interferences among $p_1, p_2, \ldots, p_m$, and the paths have length $l_0/m = O(|s_Y|/m) = O(|s_Y|^{1/3})$. Hence the total cost charged for the interferences is $O(|s_Y|^{4/3})$. □

Hence at the end of the analysis (i.e., when Mgreedy 1 terminates), the total length of current paths and cycles is $O(|s_Y|^{4/3})$. If some cycles (representing self-overlapping

strings) exist at this moment, we need to open these cycles before we can calculate the length of the superstring produced by Mgreedy 1. Opening a cycle will at most increase its length by a factor of 6. Thus, the length of the superstring produced by Mgreedy 1 for Y is $O(|s_Y|^{4/3}) = O(n^{4/3})$. This completes the proof of Theorem 7. □

If the number of negative strings is bounded by some constant, we can show that our algorithm in fact achieves linear approximation.

Corollary 17. *Given (P, N), where no string in P is a substring of a string in N and $|N| \leqslant c$, c is a constant, then the above algorithm returns a consistent superstring of length $O(n)$.*

Proof. (*Idea*). Observe that using each negative string in N, a path can only interfere some other path just once. Thus each negative string can cause no more than $O(n)$ extra cost. Hence the resulting superstring is of length $O(n)$ following the proof of Theorem 7. □

5. Concluding remarks

We have given polynomial-time linear approximation algorithms for two special cases of the shortest consistent superstring problem. It still remains open if a polynomial-time linear approximation algorithm exists for the general case. We suspect that our $O(n^{4/3})$ upper bound on the performance of a greedy algorithm, in the special case when no negative strings contain positive strings, can be improved to $O(n)$.

6. Acknowledgement

We have enjoyed and benefited a lot from working with A. Blum, J. Tromp and M. Yannakakis on the shortest common superstring problem. We are also grateful to the referees for their valuable comments and suggestions.

References

[1] A. Blum, T. Jiang, M. Li, J. Tromp and M. Yannakakis, Linear approximation of shortest superstrings, in: *Proc. 23rd ACM Symp. on Theory of Computing* (1991) 328–336; also to appear in *J. ACM*.

[2] R. Drmanac and C. Crkvenjakov, Sequencing by hybridization (SBH) with oligonucleotide probes as an integral approach for the analysis of complex genomes, *Internat. J. Genomic Res.* 1-1, 1992, 59–79.

[3] J. Gallant, D. Maier and J. Storer, On finding minimal length superstring, *J. Comput. System Sci.* **20**, 1980, 50–58.

[4] M. Garey and D. Johnson, *Computers and Intractability* (Freeman, New York, 1979).

[5] A. Hoffman, On simple linear programming problems, in: V. Klee, ed., *Convexity: Proc. of Symposia in Pure Mathematics, Vol. 7* (AMS, Providence, RI, 1963).
[6] T. Jiang and M. Li, DNA sequencing and string learning, submitted for publication, 1993.
[7] T. Jiang and M. Li, On the complexity of learning strings and sequences, in: *Proc. 4th Workshop on Computational Learning* (1991) 367–371; *Theoret. Comput. Sci.* **119** (1993) 363–371.
[8] A. Lesk (ed.) *Computational Molecular Biology, Sources and Methods for Sequence Analysis* (Oxford University Press, Oxford 1988).
[9] M. Li, Towards a DNA sequencing theory, in: *Proc. 31st IEEE Symp. on Foundations of Computer Science* (1990) 125–134.
[10] P. Pevzner and R. Lipshutz, Towards DNA sequencing by hybridization, manuscript, 1993.
[11] C. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity* (Prentice-Hall, Englewood Cliffs, NJ, 1982).
[12] H. Peltola, H. Soderlund, J. Tarhio and E. Ukkonen, Algorithms for some string matching problems arising in molecular genetics, *Inform. Process.* 83 *(Proc. IFIP Congress)* (1983) 53–64.
[13] J. Storer, *Data Compression: Methods and Theory* (Computer Science Press, Rockville, MD, 1988).
[14] J. Tarhio and E. Ukkonen, A greedy approximation algorithm for constructing shortest common superstrings, *Theoret. Comput. Sci.* **57** (1988) 131–145.
[15] J. Turner, Approximation algorithms for the shortest common superstring problem, *Inform. and Comput.* **83** (1989) 1–20.
[16] L.G. Valiant, A theory of the learnable, *Comm. ACM* **27**(11) (1984) 1134–1142.